THE INTEGRAL PERSONAL COMPUTER

DRIVER WRITER'S REFERENCE MANUAL

November 1984

Revised July 1986

Hewlett-Packard Corvallis Workstation Operation

CONTENTS

1. Introduction

The Hewlett-Packard Integral Personal Computer (IPC) has a Read-Only Memory (ROM) based Operating System (OS). The OS is derived from the popular UNIX* Operating System originally designed at Bell Laboratories. In particular the current IPC OS is derived from the System V release from Bell Labs.[1] Hewlett-Packard supports an internal standard based on System V called HP-UX.

The IPC is a very flexible machine because it supports the addition of new drivers to its OS. The **dload** utility, described in this document, can allocate memory in the IPC, and load a user designed driver into the IPC. The driver could control custom hardware attached to the IPC, or could provide new OS services as required.

This document is designed to aid the programmer who wants to enhance the IPC Operating System. Particular attention is devoted to the task of writing and loading an HP-UX driver. This document describes the most important interactions between a driver and the IPC HP-UX Operating System. It is assumed that the reader has some knowledge of the C programming language and of HP-UX utilities.

Warning! Drivers and operating system enhancements are closely tied to the internal services provided by the operating system. Thus it may be necessary to recompile, or modify, a driver in order to migrate it to a different version of the IPC OS. The internal OS services *may not* maintain backward compatibility. This document and associated disc provide a driver loading utility, **dload**. **dload** is designed to minimize the possible work in migrating a driver to a new OS version. It is possible that the use of **dload** will allow a driver to work unchanged on multiple versions of the IPC HP-UX OS, yet this is *not* guaranteed.

Author's Note: This is *not* a comprehensive treatment of writing drivers for the IPC HP-UX Operating System. Nor does the author know of any book or document that does real justice to the details of driver writing. It is our sincere desire to continually improve this document. Thus if your needs or questions are not answered by a thorough use of this document, then contact your local Hewlett-Packard support organization to obtain further details.

---

* UNIX is a Trademark of Bell Laboratories

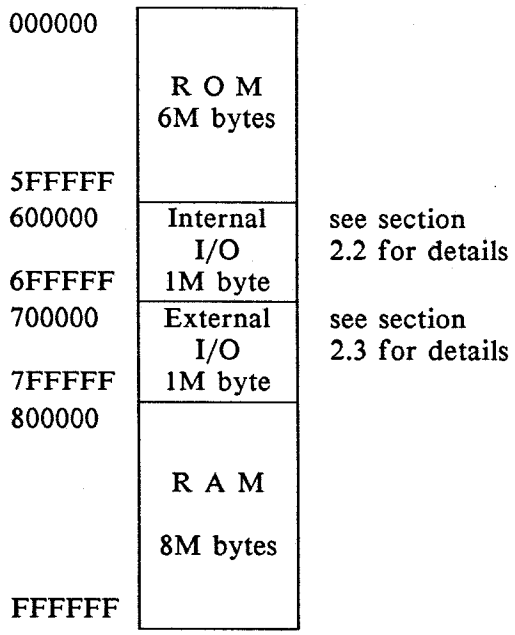
1. The current IPC OS is Release 5.0, as displayed in the copyright window on power up. This document has been revised to conform to OS Release 5.0. Any references to the previously released OS Release 1.0 will be explicitly noted.

2. Hardware Addressing

This section describes the hardware address characteristics of the Integral Personal Computer necessary to implement an HP-UX driver for the IPC.

2.1 The Memory Map

The IPC memory map is broken into three functional areas: the ROM, RAM and I/O areas. The I/O area is subdivided into the Internal I/O area and the External I/O area. These areas are shown in the diagram below:



The HP-UX OS Release 5.0 ROM occupies the first 512K bytes of the ROM address space. This leaves 5 1/2 Megabytes of ROM address space available. The IPC ROM disc driver scans the remaining ROM space looking for plug-in ROM cards. If an appropriately structured ROM card is plugged in, the ROM files found will be included in the **/rom** directory after power up. Currently there are two types of ROM cards that work in the IPC: the HP82968A EPROM/ROM Board and the HP82971A EPROM/ROM Module. HP82968A is designed to fit in the OS package module, while HP82971A EPROM/ROM Module is designed to plug into any of the I/O slots in the IPC. Refer to the HP82970A EPROM/ROM Software Development Tools Users Guide for complete details about creating software for ROM cards.

2.2 Internal I/O Addresses

The Internal I/O space provides access to the built-in peripherals and other features of the IPC. This area is allocated as shown below.

600000	Memory Mapper	- see section 2.4
610000	Disc Interface	- Western Digital 2797
620000	Graphics Processing IC	- HP Proprietary
630000	HPIB Interface	- Texas Instruments 9914
640000	Real Time Clock	- National MM58167A
650000	Printer Control	- HP Proprietary
660000	Keyboard Control	- HP Proprietary
670000	Speaker Control	- National COP 452
680000	Reserved - 512 K bytes	
6FFFFF		

2.3 External I/O Addresses

The external I/O area is allocated as shown below. Up to two bus expanders may be plugged into the IPC. Each bus expander takes up a port in the IPC, and provide five ports each. Up to ten ports are thus available. The IO slot marked A is searched first by the OS for an expander, then the B slot is checked.[2] This ordering is important in determining the port mapping of expanders if there are two bus expanders plugged into the system. If there is only one bus expander plugged into the system, then it will be mapped into ports 18-22 regardless of the slot in which it is found.

Port #	Address	I/O Slot	Location
16	700000	A	Mainframe I/O Ports
17	710000	B	
18	720000	A1 or B1	First Bus Expander
19	730000	A2 or B2	
20	740000	A3 or B3	
21	750000	A4 or B4	
22	760000	A5 or B5	
23	Reserved		
24			
25			
26	7A0000	B1	Second Bus Expander
27	7B0000	B2	
28	7C0000	B3	
29	7D0000	B4	
30	7E0000	B5	
31	Reserved		

2.4 Identifying I/O Cards

Each I/O card for the IPC contains an identifying code. This enables software to poll I/O memory addresses described in the previous section in order to determine the location and type of all I/O cards plugged into the IPC.

The card ID is contained in the least significant nibble of the first odd addressed byte of the appropriate I/O memory space. For example, consider an I/O card plugged into slot A in the IPC. Slot A is mapped to address 700000, (see the table in the previous section). Thus the ID nibble is found in the least significant nibble at address 700001.

However a nibble can only represent sixteen different I/O cards. We do not want to limit the IPC to using less than sixteen distinguishable I/O cards. So if the ID nibble

2. Note that this mapping algorithm differs from that used by OS Release 1.0. Release 1.0 always mapped ports 18-22 to an expander plugged into slot A, and an expander plugged into slot B always mapped to ports 26-30.

contains a zero, then the type of I/O card is contained in the extended ID byte, found at the next odd addressed byte. Thus if an I/O card plugged into slot A, contained a zero in the least significant nibble at address 700001, then the extended ID byte would be found at address 700003.

The following table lists the current ID codes used for the IPC I/O cards.

Primary ID Nibble Assignments

ID nibble	Card type
0	Reference Extended ID Byte
1	HPIL Interface (82924A)
2	RS-232 Interface (82919A) and Current Loop Interface (82920A)
3	GPIO Interface (82923A)
4	BCD Interface (82922A)
5	300/1200 BPS Modem (82915A)
6	Reserved
7	HPIB Interface (82998A)
8	Reserved
9	Reserved
10	Reserved
11	1 MByte RAM Module (82916A)
12	512 KByte RAM Module (82927A)
13	256 KByte RAM Module (82925A)
14	Reserved
15	Reserved

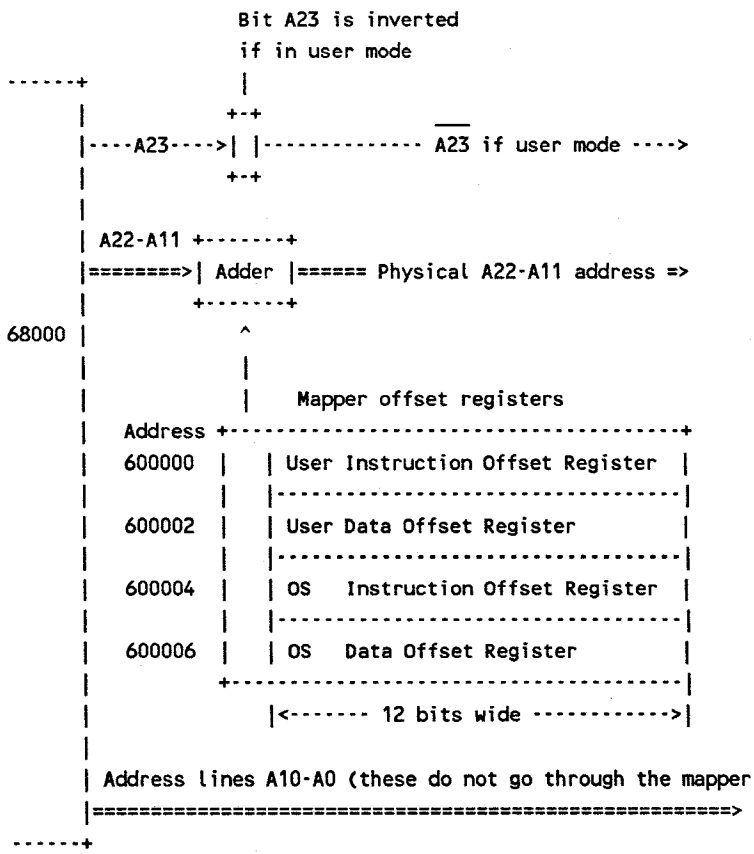
Extended ID Byte Assignments

ID Byte	Card type
0 - 31	Reserved
32	Bus Expander, not powered up (82904A)
33 - 47	Reserved
48	Bus Expander, powered up (82904A)
49 - 252	Reserved
253 - 254	open for proprietary use
255	Hewlett-Packard Testing Tool

The extended ID Bytes, 253 and 254, may be used by anybody developing a card that is intended for proprietary use only. Cards designed with proprietary ID bytes should not be designed for use by general IPC users. If you intend to design a card for general IPC use, then you should contact Hewlett-Packard in order to be assigned one of the reserved ID codes. The above tables list only ID codes used for current Hewlett-Packard products. There are cards made by third party vendors for general IPC use that are not listed in the above tables. Identifying codes can be assigned to a proposed a card in order to ensure that all known cards with controlled ID codes will be easily distinguished.

2.5 Memory Mapping

The IPC contains simple memory mapping hardware. This hardware can map addresses output by the 68000 by adding an offset to the address. The memory mapper is composed of four 12-bit registers and a 12-bit adder. The diagram shown below shows the basic structure of the memory mapper.



The mapper is activated when an address output by the 68000 processor has the "high bit" on. These addresses are in the range of 800000-FFFFFF hex. Thus the mapper only maps addresses which are in the RAM area. Accesses made to the ROM or the I/O areas are NOT mapped.

The mapper operates in two basic modes corresponding to the state of the 68000 processor. When the processor is in supervisor state, the mapper adds the OS Instruction offset to addresses output for instruction fetches. The mapper adds the OS Data offset to addresses used for supervisor state reads and writes of data. The User Instruction and User Data offsets are used in a similar manner when the 68000 is in user mode.

When the 68000 is in user mode the mapper takes a special action. This action is to invert the "high bit" of addresses output by the 68000, bit A23. This has the effect of adding 800000 to addresses issued in user mode. This is necessary to shift the process address space from address 000000-7FFFFF to 800000-FFFFFF. In this way the process address space is swapped from the ROM and I/O areas to the RAM area. The 12-bit offset is then capable of mapping the process anywhere in the 8 megabyte RAM area.

NOTE: Because bit A23 is inverted by the memory mapper for user processes, it must be "pre-inverted" if a process wishes to access the I/O or ROM areas directly. For

example, suppose a program wished to access the Graphics Processing Unit chip directly. The GPU is located at physical address 620000. To access the chip, the process should try to access address 620000+800000, or E20000. Then when the mapper inverts the bit A23, the actual address presented to the adder is 620000, which will not be mapped.

2.6 I/O Interrupt Handling

The IPC's 68000 fields interrupts on priority levels 1 through 7. Cards plugged into the external I/O ports have access to interrupt request lines for levels 3 through 6. When the 68000 processes an I/O interrupt, it stacks its PC and status register on the OS stack. Then it fetches an interrupt vector from the OS ROMs. The vector points to a high RAM location which has been initialized to a jump to the interrupt polling routine in the OS. The table below shows the addresses of the jumps in high RAM. The name *genpol* is the name of the polling routine in the OS.

Exception Vectors (in ROM)		Exception Jump Table in high RAM.
Level 1 int	-> F1FFB8	jmp _genpol
Level 2 int	-> F1FFB2	jmp _genpol
Level 3 int	-> F1FFAC	jmp _genpol
Level 4 int	-> F1FFA6	jmp _genpol
Level 5 int	-> F1FFA0	jmp _genpol
Level 6 int	-> F1FF9A	jmp _genpol
Level 7 int	-> F1FF94	jmp _lev7int

Each of the vector numbers corresponds to one of the processor priority levels. When an interrupt is serviced at a particular level, all interrupts at levels less than or equal to the processor level are masked. The processor only responds to interrupts at priority levels greater than the current processor level.

When the processor priority is set to level 7, however, level 7 interrupts are still allowed. The 68000 has this feature to provide a non-maskable interrupt facility. Thus the Keyboard Reset Interrupt cannot be masked by setting the processor priority to level 7.

When handling an interrupt on levels 3 through 6, the **genpol** routine scans a table of service routine addresses for the level. The **genpol** routine calls each interrupt service routine in the table. The interrupt service routine determines if the device it controls is requesting service. If so, then the service routine handles the interrupt, and then returns a zero status. If the device is not requesting service, then the service routine returns a non-zero status.

A sample interrupt service routine is shown below.

```
/*
 * This is a generic interrupt service routine.  The routine
 * checks the device it controls, and if the device is
 * requesting service, then handle the interrupt.
 * The interrupt level is passed to the service routine.
 */
#define I_HANDLED_IT    0       /* was interrupting status   */
#define NOT_ME          1       /* not interrupting status   */

int
example_isr( level )
int level;      /* current interrupt priority level     */
{
        /*
         * if the device is not requesting service by
         * interrupting, return to the polling routine
         * with the "not me" status.
         */
        if (!(<TEST FOR INTERRUPT REQUEST>))
                return(NOT_ME);
        /*
         * Handle interrupt.... then return
         */
        return(I_HANDLED_IT);   /* tell polling routine to stop polling */

} /* end of interrupt service routine */
```

The OS contains a RAM based table of interrupt service routine addresses for interrupt levels 3 through 6. Each level has eight entries for service routines. At power on the table is configured as shown below.

Interrupt Level	Contents of Poll Table							
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	Internal Disc	Serial Card	0	0	0	0	0	0
6	Interval Timer	0	0	0	0	0	0	0

The entries with zero in the table indicate available slots. When the interrupt poller scans the table, it will call the routine addresses until a service routine indicates it handled the interrupt, or when a zero address is encountered.

The **dload** utility, described later in this document, will load the address of your interrupt routine into the proper slot in this table. For example, if your code contains the lines:

```
char *patch[] = {
        "interrupt.5<-myinterrupt",
        0
};
```

then **dload** would load the address of your routine, **myinterrupt()**, into the first available slot in the level 5 set of vectors. In this example the first available slot immediately follows the interrupt vector for the serial interface driver. If there are no available slots in the vector table at the proper level then **dload** will issue an error message.

Note that the level 5 interrupt vector for the serial interface driver will be placed in the table regardless of the presence of a serial interface at power up. Note also that the modem driver, shipped on the system disc, makes an entry in the level 3 interrupt set of vectors. Both the modem driver and the serial driver require only one entry in the interrupt table regardless of the number of serial or modem cards plugged into the system. Note other loadable drivers may make entries into the interrupt table. For example, IPC Technical BASIC includes drivers for other interface cards, (like the HP-IL interface) which may require entries in the interrupt service routine table. The IPC Device I/O Library (DIL) HPIB driver makes an entry for a level 3 interrupt.

The remaining interrupt levels are used as shown below.

Level 1: Real time clock interrupt (every 100 ms)

Level 2: Keyboard/Mouse interrupt

Level 7: Control-Shift-Reset from keyboard (not maskable)

Warning!: Interrupt service routines should be written with great care. Many actions which are otherwise quite legal must be avoided when executing at an interrupt level. For example, the **sleep()** OS routine should not be called from an interrupt routine, because the interrupt is an asynchronous event with respect to process operation. When the **sleep()** call is made, the interrupt service routine most likely will put the wrong process to sleep! For this reason, the interrupt service routine also should not call any OS routines which might call **sleep()**. Some examples of OS routines that may call **sleep()** are: alloc(), dopen(), free(), openp(), physio(), plock(), ptrace(), sigpause(), wait().

Modifications of global data structures by an interrupt service routine can be dangerous. In particular, modification of the elements of the **u** structure should be avoided. From the OS's program level point of view, an interrupt routine which modifies global data structures will produce "phantom" changes, and will be difficult to track down.

3. General Driver Information

3.1 Introduction

The IPC operating system supports two kinds of drivers. These drivers are called *block drivers* and *character drivers*. A block driver deals with a block of bytes for each operation it is requested to perform. A character driver, as its name suggests, deals with data a character at a time. The IPC supports the addition both kinds of drivers to the system via the driver loader described in the section entitled "The dload Driver Loader Program". Appendix II describes a sample character driver.

3.2 Device Files

An application program communicates with a driver through a file traditionally located in the **/dev** directory. The files in **/dev** are known as *special files*. The files are created with special attributes which causes the OS to deal with them differently from normal data files. The files can be created using the **mknod(1)** utility or programmatically using the **mknod(2)** kernel call. The parameters provided to **mknod** define which block or character driver will be associated with the special file. Please refer to the section titled "Driver Special Files" for more information on the creation of special files.

When a program opens a special file, the HP-UX operating system calls the **open()** routine of the associated driver. The driver can then perform any error testing and initialization it needs to do. If desired, the driver can simply return, i.e. the **open()** routine can be null.

When a program closes the special file the HP-UX *may* call the driver's close routine. The call to the driver's close routine is made *only on the last close* of the driver. The OS makes the decision on whether to call the close routine by keeping a count of the number of times the special file has been opened. Each time the special file is closed, the count is decremented. When the count goes to zero, the close routine of the driver is called.

When a program reads from or writes to a special file, the OS examines what kind of driver is being used. If the driver is a character device driver then OS calls the driver's read or write entry points. If the driver is a block device driver then the OS calls the driver's strategy routine.

3.3 Error Handling

In testing for errors, the IPC operating system does not examine any values returned by any driver routines. Instead, it will examine a global variable **u.u_error**. The **u** is the name of a structure which is associated with each process. The **u_error** is an unsigned char variable contained in this structure. Should the driver detect an error, it should write an error number into this variable. The errors to use are defined in the header file **/usr/include/sys/user.h**. You will need to include **sys/user.h** in order to setup the definition of the **u** structure for the compiler.

When a driver has returned to the OS, the OS will examine the **u.u_error** variable before it returns control to the program. Should the OS notice that a driver has written an error number to the **u.u_error** variable, it will return the error number to the program. The number is written to the **errno** variable which the program may examine.

3.4 Character Device Drivers

A character driver connects with the HP-UX operating system through five entry points. These entry points, listed below, are fully described in the sections following.

open - called when the driver's device file is opened.

close - called on the last close of the driver's device file.

read - called by a process to read data from the driver.

write - called by a process to write data to the driver.

ioctl - called by a process to perform special driver-dependent functions.

3.5 Block Device Drivers

A block driver connects with the HP-UX operating system through three entry points. These entry points, listed below, are fully described in the following pages.

open - called when the driver's device file is opened.

close - called on the last close of the driver's device file.

strategy - called to request a block data transfer.

If the driver supports it, the HP-UX operating system will allow a block device to be mounted to a target directory as a disc. The IPC contains two drivers which emulate discs - the RAM disc and the ROM disc. The RAM disc driver uses allocatable system RAM to provide a virtual disc. The ROM disc driver converts the files found in the any ROM modules into a virtual disc. Three more disc drivers are built into the IPC. These disc drivers support the internal disc, and access to Amigo and Subset-80 protocol HPIB discs. The Amigo and Subset-80 protocols are used within Hewlett-Packard for disc communication to proprietary disc products. The Amigo and Subset-80 disc drivers are designed to work with disc drives connected to the built-in HPIB interface.

4. Driver Entry Points

The following pages describe the driver's entry points. These routines form the interface between the OS and a driver.

4.1 Open

```
open (dev,oflag);
dev_t  dev;
int    oflag;
```

When a user process issues an **open** or **creat** call to a special file in **/dev**, the OS will process the **open** or **creat**, and then call the driver associated with the special file. Note that the syntax shown above is different than the syntax of the open call as issued by the user program. The HP-UX OS will check a number of things before issuing the **open** call to a driver. In particular, the IPC OS will:

Create a file descriptor in the OS file array, and save the value of the **flag** parameter passed in with the user's open call. This parameter will be copied into **u.u_fmode** before each read and write call to a driver. The flag is also passed in line for any ioctl calls made to a driver. Thus, although the driver is free to examine the **oflag** parameter at open time, it does *not* have to save this parameter.

Use the saved **oflag** to check subsequent read and write calls made by a particular process to make sure that the driver has been opened for that action. (This will catch writes to a driver which has been opened for read-only operation, and vice-versa).

The **oflag** parameter may have the following bits set:

name	value	description
FREAD	0x01	open for reading
FWRITE	0x02	open for writing
FNDELAY	0x04	non-blocking open

The non-blocking open option is specified by user programs which desire the driver to return if no bytes are available to be read. FREAD, FWRITE, and FNDELAY are all defined in the file **/usr/include/sys/file.h**.

4.2 Close

SYNOPSIS

```
close (dev)
dev_t       dev;
```

DESCRIPTION

close() is called to do any cleanup work that needs to be done by the driver. Although the OS calls the driver's **open** entry point *each time* a process opens a driver, the **close** call will be made to a driver *only by the last* process to close the driver. The OS makes the decision on whether to call the **close** routine by keeping a count of the number of times the special file associated with the driver has been opened. Each time the file is closed, the count is decremented. When the count goes to zero, the **close** routine of the driver is called.

4.3 Read (Character Device Drivers)

SYNOPSIS

```
read (dev)
dev_t dev;
```

DESCRIPTION

read() is used to transfer bytes from the device to a buffer specified by the caller. The *user structure* for the current process contains the parameters passed in by the user program pertaining to the read call. The user structure is defined in **/usr/include/sys/user.h**. Should one driver call another driver's **read** entry point, the calling driver should set up the parameters, or see the **dread()** utility routine documented in section 6. The parameters a driver needs to examine during a read call are:

u.u_base Contains the base address of the read buffer.
u.u_count Bytes remaining in the read request.
u.u_offset Offset in the file for I/O.
u.u_segflg Memory map flag:
0 => user data map
1 => kernel map
2 => user instruction map
u.u_fmode The FNDELAY bit will be set if the read is non-blocking.

If the **read** is done using blocking mode, then the calling process will be put to sleep until the requested number of bytes is read. If the driver supports non-blocking reads and the read is done using non-blocking mode (see the open call) then the driver should return immediately with any bytes that may be available.

The driver should decrement **u.u_count** by the number of bytes written to the user's buffer. Under no circumstances should the driver return more bytes than were requested by the program.

The **passc()** routine is very useful for writing bytes to the user's buffer from the driver. The routine updates **u.u_count** and **u.u_base** for each byte written.

4.4 Write (Character Device Drivers)

SYNOPSIS

```
write (dev)
dev_t dev;
```

DESCRIPTION

write() is used to transfer bytes from the caller to a device controlled by the driver. The *user structure* for the current process contains the parameters passed in by the user program pertaining to the write call. The **user** structure is defined in **/usr/include/sys/user.h**. Should one driver call another driver's **write** entry point, the calling driver should set up the parameters, or see the **dwrite()** utility routine documented in section 6. The parameters a driver needs to examine during a write call are:

u.u_base	Contains the base address of the write buffer.
u.u_count	Bytes remaining in the write request.
u.u_offset	Offset in the file for I/O (may be ignored by drivers).
u.u_segflg	Memory map flag: 0 => user data map 1 => kernel map 2 => user instruction map

4.5 Ioctl (Character Device Drivers)

SYNOPSIS

```
ioctl (dev, com, argp)
dev_t dev;
int com;
char *argp;
```

DESCRIPTION

The **ioctl** (I/O control) command is used to send device specific commands to a driver. The **com** parameter defines the command, and the **argp** pointer points to any arguments required by the command. The **ioctl** commands are driver specific. Each driver generally implements a distinct set of **ioctl** commands. The documentation for drivers is generally available as manual pages in section 4 of the HP-UX system documentation.

The driver's ioctl handler is generally structured as shown in the skeleton C routine below. The example shows how a driver can interact with a user program in setting parameters in the driver. The example driver contains a structure with operating parameter information. The GET_PARMS command causes the driver to write its parameter structure to the program issuing the ioctl. The SET_PARMS command causes the driver to read a new parameter data into its local structure.

The example shows the driver including the **user.h** file. This file defines the per process user structure **u**, and errors such as the EINVAL (invalid argument) error used in the example routine. Other include files, such as **sys/param.h**, **sys/ino.h**, **sys/inode.h**, and **sys/dir.h** are required to provide definitions used by the **user.h** file.

```
/*
 * skeleton driver ioctl handler example
 */
#include <sys/param.h>
#include <sys/dir.h>
#include <sys/ino.h>
#include <sys/inode.h>
#include <sys/user.h>
#include "ioctldefs.h"

struct my_data md;
  .
  .
  .
driver_ioctl( dev, command, argp)
dev_t dev;
int command;
char *argp;
{
        switch( command ) {
        case GET_PARMS:
                /*
                 * read the new parameters from the caller.
                 */
                copyout( &md, argp, sizeof(md) );
                return;
        case SET_PARMS
                /*
                 * write the present parameters to the caller.
                 */
                copyin( argp, &md, sizeof(md) );
                return;
        default:
                /*
                 * the caller passed an undefined command
                 */
                u.u_error = EINVAL;
                return;
        }
}
```

The **command** parameter is traditionally structured as a character which has been left shifted 8 bits, then or'ed with a number. Also traditionally each left shifted character corresponds to one specific driver in the system. For example, the TCGETA (Terminal Command Get Attributes) ioctl command is defined in **termio.h** to be **(('T'<<8)|1)**. The 'T' character corresponds to the tty driver. In a similar manner, characters for other devices have been defined. This helps to sort out which command belongs to which driver. It also provides a measure of error checking, in that a driver can check the ioctl commands to make sure the command is one it implements. A command could be misdirected if, for example, an application program opened the wrong special file, and then issued an ioctl command. In the example ioctl handler above, the error would be caught by the default clause in the **switch** statement.

The following table lists the ioctl letter assignments:

Char	Driver
A	Alpha/Graphics window
a	common to all drivers
B	beeper driver
D	disc drivers
f	fast alpha facility
G	Graphics Processing Unit (GPU) driver
h	HPIB interface driver
k	caravan loop (HP-HIL) driver
L	printer driver
P	plotter emulator
p	generic plotter driver
R	realtime driver
s	serial interface driver
T	tty driver
w	window manager driver
z	terminal 0 driver (see terminal.h)

4.6 Strategy (Block Device Drivers)

SYNOPSIS

```
strategy (bp)
struct buf *bp;
```

DESCRIPTION

Block drivers differ from character drivers in that they do not contain individual read or write routines. Instead the driver contains a single **strategy** routine. This routine has the responsibility of reading or writing blocks of data. In the IPC, the blocks are an integer multiple of 1024 bytes.

When the OS calls the strategy routine, it passes a pointer to a buffer header. The buffer header is defined in the include file **sys/buf.h**. The table below describes the elements of interest to a block driver:

b_flags This flag contains status information about the block. In particular the flag tells whether the block is to be read or written. The flags of interest to the strategy routine are explained below.

- B_READ — If this bit is set, then the transaction is a block read. If clear, then a write is to be performed.
- B_DONE — This bit is set by the **iodone(bp)** routine for the driver. The **strategy** routine should call **iodone(bp)** when it is done with the block request given by **bp**.
- B_ERROR — This bit should be set in **b_flags** when the **strategy** routine detects an error. It may also optionally set the **b_error** variable. If it does not set the **b_error** variable, the OS will default to returning an EIO error to the process making the block request. Otherwise the error written to **b_error** will be returned to the process making the block request.
- B_PHYS — This bit if set indicates that the block transaction was made through the *raw device*, and that the address of the block is given by **b_un2.byteno**, rather than **b_blkno**.

b_dev Tells which device the block is to be read from or written to. If the block driver controls only one device this variable can be ignored.

b_count Contains the number of bytes to be read or written. This number will be an integral multiple of 1024.

b_un.b_addr This is a pointer to the address to read the block from or write the block to. The **b_un** is a union of several elements also defined for the convenience of various other OS routines. The driver need only worry about the **b_addr** element of the union.

b_blkno This element contains the block address of the block to be operated on. The block numbering starts at 0.

b_un2.byteno This element is defined if the driver is called through a *raw access* mechanism. A raw device is essentially a character special device, as opposed to a block special device. If the access is a raw access, then the **b_un2.byteno** offset is used as the disc address rather than **b_blkno**. The **byteno** offset is a byte offset, not a block offset. The **byteno** offset is restricted to be an integral multiple of a block.

b_error Errors detected by the driver are reported by setting this element to an error number which most closely approximates the error. A block driver should *not* use **u.u_error** to report the error. The OS checks **b_error** after the block driver has indicated that the I/O on a block has been done.

b_resid If the driver detects an error before transfering all the block(s) requested, then it should indicate how many bytes are left to be transfered in this variable. The driver *should clear* **b_resid** when a block has been successfully transferred.

The fragment of code below shows an outline of a strategy routine. The routine shown checks to see if the request is within the size of the device it is controlling, and returns an error if the request is out of bounds. In the example BSHIFT is defined to equal 10 by the **filsys.h** header file.

```
strategy( bp )
struct buf *bp;
{
    if (bp->b_flags & B_PHYS) {
        /*
         * form the block number from the byte offset
         */
        bp->b_un2.b_blkno = bp->b_un2.b_byteno >> BSHIFT;
        bp->b_un.b_addr = bp->b_phaddr;
    }
    /*
     * check to see if the request is out of bounds
     */
    if (bp->b_un2.b_blkno > MY_MAX_BLOCK) {
        /*
         * yes, flag an error back to the OS
         */
        bp->b_flags |= B_ERROR;
        bp->b_error = ENXIO;
        return;
    }

    /*
     * handle the request ....
     */

    /* make sure b_resid is cleared! */
    bp->b_resid = 0;
    iodone(bp);
}
```

It can be helpful to understand the relationship between the calls in a user program to **read()** and **write()** and the calls to the driver **strategy()** routine. The user program calls to **read()** and **write()** are handled by the operating system. The operating system in turn decides when to invoke the **strategy()** routine. The IPC HP-UX operating system implements what might be called a *partial synchronous file system.* A true synchronous file system would perform reads and writes at the same time a user program requested them. Traditionally UN*X has buffered read and write requests in order to obtain

overall better I/O performance. The IPC HP-UX OS implements normal buffering for **read()** calls. While for **write()** calls the OS adopts aspects of a synchronous system.

There is a one-to-one correspondence between calls in the user program to **write()** and calls to the driver **strategy()** routine. This means that a user program that writes one character at a time to a device, will cause the OS to invoke the **strategy()** routine for each character to be written. This behavior has been helpful in designing a system that permits the user to use removable media without needing to constantly mount and unmount discs or to issue **sync** commands in order to force writes to disc media. Note also that for a block device there is always a 1K byte block buffer handed to the driver strategy routine. Thus the *worst case* of single character writes to a block device actually requests the **strategy()** routine to write a 1K byte block for each user request of a single character write. Note also that the OS will read a block from the device driver before it attempts to write a portion of a 1K byte block. If the OS already has the target block in a buffer from a previous read or write, then the block saved in a buffer by the OS will satisfy the read request. This insures the integrity of the *entire* 1K byte buffer to be written, regardless of the number of characters requested to be written by the user program.

User **read()** requests are handled by the OS differently than user **write()** requests. The user **read()** requests are buffered by the OS. A buffer cache is maintained by the OS. Thus a series of single character reads from a user program will trigger an initial request for a 1K byte block read to the **strategy()** routine. Then if further read requests can be satisfied by the block retained by the OS, then the **strategy()** routine will *not* be invoked.

5. Writing a Driver

5.1 Objectives

The objective of a driver is to provide a service not found in the standard OS. The driver could be code to comminicate with a new hardware device, or it could be new code which is to replace an old driver so as to add new functionality or to fix errors. The intent of a driver writer is then either to add completely new functionality or to replace old functionality by something different and/or new.

The Integral OS supports both the addition of new drivers and the replacement of old drivers. Various tables are described below which contain addresses of driver entry points. These addresses can be modified by the driver writer via the **dload** driver loader utility. If a table entry pointing to an old driver is overwritten, then the new driver will be in control. If a new table entry is created, then the driver will be added to the system.

5.2 Places in the OS which can be patched.

The IPC OS has various tables which can be patched by a driver with the aid of the driver loader program. A patch is essentially effected when a vector containing the address of an OS routine is changed to contain the address of a routine written as an OS enhancement. These tables are described below:

5.2.1 The cdevsw Table

The **cdevsw** table contains the addresses of the entry points for character device drivers in the IPC OS. Each table entry contains five addresses. These addresses correspond to a driver's **open**, **close**, **read**, **write**, and **ioctl** entry points. To make a character driver available for use by the OS, its entry points must be placed in the **cdevsw** table by the driver loader.

The IPC's **cdevsw** table has 33 table entries. The IPC uses entries 0-24, or 25 total, of the entries available. The remaining 8 entries are available for drivers added by the driver loader.

The **dload** program can either add a new driver to one of the 8 unused entries, or it can overwrite one of the old entry points pointing to a driver.

The **dload** program, when it finds that a new driver is to be added into the **cdevsw** table, scans the 8 available entries for a position that is unused. The 8 entries are set up at power on to call a particular error routine for all five (open, close, etc) entry points for each driver position in the table. The **dload** program scans each driver position to look for an available slot. When the **dload** program looks in a position, if all five entries still point to the error routine, the slot is assumed to be unused. The **dload** program will then load the driver and write the driver's entry points into the slot in the table. The entry points into the driver are defined by the driver's patch entry. The patch entry is described below.

5.2.2 The bdevsw Table

The **bdevsw** table contains the addresses of the entry points for block device drivers in the IPC OS. Each table entry contains four addresses. These addresses correspond to a driver's open, close, strategy and local buffer entry points. To make a block driver available for use by the OS, its entry points must be placed in the **bdevsw** table by the driver loader.

The IPC's **bdevsw** table has 14 table entries. The IPC uses entries 0-5, or 6 total, of the entries available. This leaves 8 entries available for driver's added by the driver loader.

The **dload** program can either add a new driver to one of the 8 reserved entries, or it can overwrite one of the old entry points pointing to a driver.

The **dload** program, when it finds that a driver is to be loaded into the **bdevsw** table, scans the 8 available entries for a position that is unused. The 8 entries are set up at power on to call a particular error routine for all three (open, close, etc) entry points for each driver position in the table. The **dload** program scans each driver position to look for an available slot. When the **dload** program looks in a position, if all three entries still point to the error routine, the slot is assumed to be unused. The **dload** program will then load the driver and write the driver's entry points into the slot in the table. The entry points into the driver are defined by the driver's patch entry. The patch entry is described below.

5.2.3 The Interrupt Service Table

The interrupt service routine table contains addresses of interrupt service routines for interrupt levels 3-6. The table contains groups of eight addresses, one group for each interrupt level. An entry of zero indicates the end of the table. The driver loader can add its entry point to the interrupt service table at the request of a driver module.

5.2.4 The Jump Table

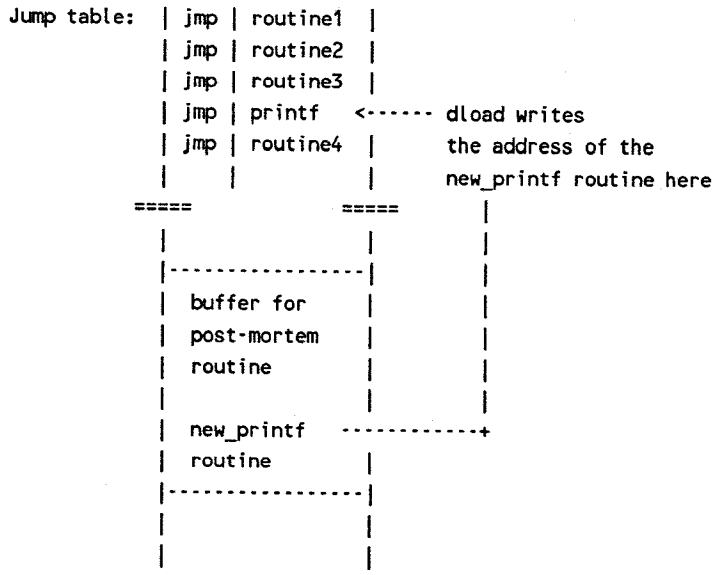
At power on the OS creates a jump table in RAM containing an array of jumps to various routines in the OS ROM. Calls by routines in ROM to other routines sometimes are made indirectly by jumping to the jump table, which in turn jumps to the actual routine in ROM. A driver can take over a jump table entry by asking the driver loader to write its entry point to the jump table entry. In this way a driver can *patch out* a routine in ROM by substituting code in the driver.

The patches to the jump table require that an offset of two be added to the target of the patch. The offset moves the target of the patch past the two byte 68000 jump opcode. For instance, the post-mortem printout facility patches the internal OS printf jump table entry, **jt_printf**. The patch entry is:

"jt_printf+2<-new_printf"

for the patch.[3] The diagram below shows how the **dload** program writes the address for the **new_printf()** routine into the jump table entry for **jt_printf**:

3. Note that the syntax for this expression "jt_printf+2<-new_printf" is parsed by the dload program itself. This syntax is not a part of the C language. The syntax is described in the following section on the dload Driver Program



5.3 Compiling and Linking a Driver

Compiling a driver module is different from compiling a normal user program. The driver, when it is loaded into its buffer in OS RAM, will be run in supervisor mode. The following two lines show how an example driver would be compiled and linked.

```
cc -c -W0,-! example.c
ld -r -x -d -o example example.o
```

In the example, the driver code is in a module called **example.c**. The module is compiled into an object module called **example.o**. The **example.o** module is then linked to form the loadable driver module example. The **-c** switch in the **cc** line tells the compiler to just compile the **example.c** module to assembly code, then assemble the code, and go no further. This produces the **example.o** module, which is input to the loader in the next line.

What does the **-W0,-!** switch in the **cc** command line do?[4] This command line option turns off the generation of stack checking code by the compiler. The compiler normally inserts special code into a program at the entry into each subroutine to check that sufficient stack exists for the execution of this routine. This code is a **trap #2** 68000 instruction, which asks the OS to check the stack, and allocate more stack space if necessary. *If the trap #2 instructions are present in a driver module*, and the module is run as part of the OS, *the system will crash!!* Therefore it is *mandatory* to use the **-W0,-!** command line option in the **cc** invocation.

The **ld** command line contains several switch options. The **-r** option tells the loader to generate relocatable code. This allows the **dload** driver loader program to relocate the driver into the buffer allocated by the OS. The **-x** switch eliminates local symbols

4. If you are using Release 1.0.0 of the C compiler then the cc command should be "cc -c -! example.c". Release 1.0.0 of the C compiler does not implement the newer -W command line option.

generated by the compiler from the symbol table. This will mean less work for the **dload** program when it loads the driver into the OS. The **-d** option tells the loader to allocate the unitialized variables (BSS variables). Normally the use of the **-r** option tells the loader to *not* allocate the unitialized variables, and so the **-d** switch turns this back on.

The resulting driver module, called **example**, is ready to be loaded by the **dload** program. The **dload** program can now copy it into the OS RAM, and fix up its references to the OS.

5.4 The dload Driver Loader Program

The **dload** driver loader program is a utility which does the work of loading a driver into the OS. The **dload** program allocates a buffer in the OS for the driver, and then copies the driver into the buffer. Then **dload** writes addresses which point to the driver's entry points into the OS system tables. Finally **dload** will optionally create a device name in the /**dev** directory for the driver.

In order to connect the driver to the OS, **dload** needs to know the addresses of the driver's entry points. The loader also needs to know how the entry points are to be added to the OS. To tell the loader this information, a patch table must be included in the driver. To illustrate how a patch table is structured, the patch entry for the example **driver1** is shown below.

```
/*
 * the patch entries are scanned by dload as it
 * loads the driver into the OS.  The patch entries specify
 * which routines in the driver are to be inserted, in this case,
 * into the cdevsw table in the OS.
 */
char *patch[] = {
        "cdevsw.name:driver1",
        "cdevsw.open<-driver1_open",
        "cdevsw.close<-driver1_close",
        "cdevsw.read<-driver1_read",
        "cdevsw.write<-driver1_write",
        "cdevsw.ioctl<-driver1_ioctl",
        0,
        };
```

The patch table is an array of pointers to strings contained in the driver. Each string can define one patch to the OS. In the **driver1**, the first line tells **dload** to create a device file in /**dev** called /**dev/driver1** when the driver is loaded. The next line tells **dload** that the driver's **driver1_open()** routine is to be added to the OS **cdevsw** table. The **cdevsw** table contains addresses of entry points for a driver's open, close, read, write, and ioctl routines. The **cdevsw.open** parameter means that the **driver1_open()** address is to be added to the **cdevsw** table's open address position. Similarly, the address of the **driver1_close()** routine is entered into the **cdevsw** table's close entry, and so on.

The available entries in the **cdevsw** table are entries 25 through 32. The driver loader scans these entries to find an available entry. When it finds one, it remembers the entry number, and then fills in the entrie's positions with addresses of routines in the driver as instructed by the patch entry.

The patch table must be called **patch** and be located somewhere in the driver. The possible entries for the patch are given by the BNF diagram below:

```
<PatchEntry> ::= <DriverName> | <Patch> | <WindowType> | <WindowMajorNum>
<DriverName> ::= <Dname>:<devicename>
<Dname>      ::= cdevsw[(<position>)].name
               | bdevsw[(<position>)].name
               | name
<devicename> ::= Name of special file to be created in /dev
<Patch>      ::= <Target>-><Source>
<Target>     ::= <SymbolName>
                 | <SymbolName>+<Offset>
                 | cdevsw[(<position>)].<c_entry>
                 | bdevsw[(<position>)].<b_entry>
                 | interrupt.<i_entry>
                 | wroutines.<w_entry>
<SymbolName> ::= Name of a symbol in the OS
<Offset>     ::= Decimal number offset from the symbol
<c_entry>    ::= open | close | read | write | ioctl
<b_entry>    ::= open | close | strategy
<i_entry>    ::= Decimal number in the range of 3 to 6 of the
                   interrupt level
<w_entry>    ::= size | init | show | unshow | key | dest | cork | uncork
                 | bf_wi | wi_bf | bi_bf | wi_bi | label
<Source>     ::= Name of a routine in the driver module or in the OS
<WindowType> ::= wtype:<w_type>
<w_type>     ::= Decimal number in the range 0 to 9 of the window type
<WindowMajorNum> ::= wmajor:<w_major>
<w_major>    ::= Decimal number in the range 0 to 32 of the window
                   major device number
<position>   ::= Decimal number of the position in the cdevsw or
                   bdevsw table to be taken over by a driver (optional)
```

The list below shows some example patch lines. Embedded blanks are not allowed in the patch lines. The patch entry must be called **patch**, as this name is hard-wired into the driver loader.

```
char *patch[] = {
        "jt_printf+2<-newprintf",
        "interrupt.6<-myinterrupt",
        "cdevsw.open<-myopen",
        "cdevsw(5).close<-newclose",
        0,
};
```

The line **jt_printf+2<-newprintf** is used by the post-mortem printout facility. It patches the jump table entry for printing information to the internal printer from the OS. The **+2** offset is necessary for patching jump table addresses. A 68000 jump instruction consists of two bytes for the jump opcode, followed by four bytes of absolute address. The **+2** insures that the driver loader will skip over the jump opcode and place the address of **newprintf()** into the old jump table address.

The **interrupt.6<-myinterrupt** line causes the driver loader to scan the level 6 interrupt table for an free entry. A free entry in the interrupt table is indicated by a zero. When **dload** finds a zero in the level 6 interrupt service routine table, it will enter

the address of the **myinterrupt()** routine into the table.

The **cdevsw.open<-myopen** line instructs **dload** to scan for an open slot in the **cdevsw** table. When **dload** finds a slot, it will write the address of the **myopen()** routine of the driver into the **cdevsw** table's open position.

The **dload** utility can also replace a driver in the OS. To do this it must overwrite an old address in the **cdevsw** or **bdevsw** tables. The **cdevsw(5).close<-newclose** entry instructs **dload** to write the address of the **newclose()** driver routine into the 6th slot of the **cdevsw** table in the close position. Indices into the tables begin with zero.

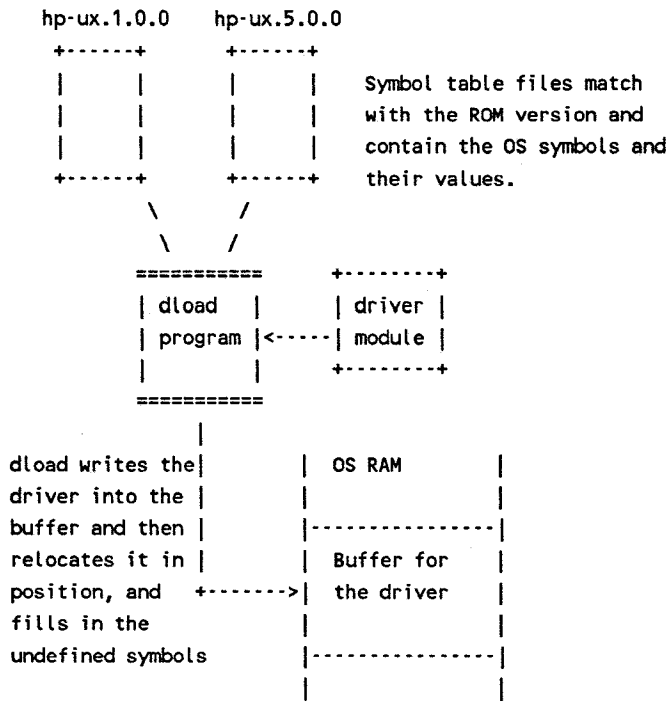
The zero entry at the end of the patch table terminates the table. The driver loader stops scanning the table when it finds the zero entry.

5.4.1 How the dload program works

When **dload** is run, it first determines the version of the OS software ROMs on which it is running. From the version it forms the name of a symbol table file which corresponds to the version of ROMs. The symbol table file name is called **hp-ux.X.Y.Z**. The suffix .X.Y.Z refers to the version of the OS ROMs. The **dload** program then attempts to find the symbol table file. The **dload** program scans the current disc, then it looks in various folders using the search path specified by the environment variable **PATH**. If the HP-UX symbol table still cannot be found, then the **dload** program asks the user to put in their system disc. For newer versions of ROMs, the HP-UX symbol table file will be shipped on the system disc for the product. The **dload** program will then read in the symbol table.

Once the symbol table is read into the **dload** program symbol table array, it is ready to read the driver. It first determines the buffer size necessary for the driver, and asks the OS to allocate space for the driver. The **dload** program then copies the driver into the buffer.

The diagram below shows the **dload** program loading in a driver.



Now the **dload** program scans the driver's symbol table to see if the driver references any OS symbols. These symbols will be undefined in the program up to now. The **dload** program looks up each undefined symbol in the driver in its array. The array has been set up previously by reading the HP-UX symbol table file, as you remember. The **dload** program then writes the address values into the driver to do the final load of the driver.

Finally the **dload** program scans the patch entries specified in the driver. For each patch entry the **dload** program makes the appropriate patch to the OS table in RAM. The **dload** program then makes any driver special files requested by the driver in the patch table.

5.4.2 dload command line options

The **dload** program has two command line options. These options take the form of standard HP-UX switches, and are:

-v The **-v** switch causes **dload** to run in *verbose* mode. The **dload** program will then printout what it is doing as it loads a driver. This can be used by a driver writer to check the operation of **dload** as it loads a prototype driver.

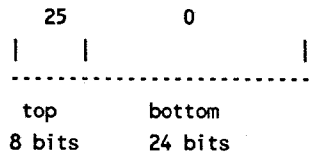
-q The **-q** switch causes **dload** to run in *quiet* mode. In quiet mode the **dload** program will only print messages when errors occur.

5.5 Driver Special Files

As described above, the driver loader provides the facility of automatically creating a special file for a driver. This is done by including a line of the form **name:mydevice** or **[cb]devsw.name:mydevice** in the patch entries. The loader will then create a special file suitable for the driver. A special file is only required when a character or block driver is being loaded. The special file will be created in the **/dev** directory.

The special file references the driver through the device number given by the driver loader when it created the special file. Device numbers are interpreted by the IPC OS as having two fields. These fields are the major number and the minor number. The major number is the top eight bits of the device number. The minor number consists of the bottom 24 bits of the device number. The major number for any driver is the entry number in the **bdevsw** table (for block drivers) or the **cdevsw** table (for character drivers). The minor device number can be any number. It is up to the driver to interpret the minor device number in whatever manner it deems appropriate. Often the minor device number determines which device on a particular hardware interface will be accessed. The major number of the device number created by the driver loader has the entry number for the appropriate table. The minor number is left as 0. If a driver requires a non-zero minor number, then the special file will need to be created by some other means, see **mknod(1)** and **mknod(2)**.

For example, suppose that the driver loader has just loaded the **driver1** example program. Suppose also that it loaded the driver into the **cdevsw** table in entry 25. When it creates the **driver1** special file in **/dev**, it will specify a device number as shown below:



The actual device number for **/dev/driver1** will then be hex 0x19000000. If you should look at the devices using the **ls -l** command, then what will be reported will be:

```
crw-rw-rw-   1 100      100       25   0 Nov 30 16:40 /dev/driver1
```

The **c** in the **crw-rw-rw-** part of the line indicates that **/dev/driver1** is a character device driver. The **25 0** fields show that the device has a major number of 25 and a minor number of 0, as expected.

5.6 Debugging Drivers

Invariably we are forced to confront our own mistakes. Finding and fixing bugs in a driver or OS enhancement requires the use of new routines. When you are executing code in the kernel (in 68000 supervisor mode) you cannot call OS entry points available to the normal user program. That means you cannot simply call **printf()** and send strings to **stdout**. Also the debugging programs, like **cdb**, are designed to work only with user programs. They will not help with kernel code. There are essentially two routines which will prove very useful in kernel debugging. Take a look at the file **/dwg/Sources/PM.src/kprintf.c**. This routine defines a scaled down version of the user level **printf()** routine. **kprintf.c** provides a routine called **new_printf()** which is callable from the kernel. **new_printf()** takes parameters in the same fashion as **printf()**. Thus if you compile **kprintf.c** into your driver module you can then call **new_printf()** to gain debugging output from your driver.

Note that **new_printf()** calls **lpi_write()** to get its job done. Another example of the use of **lpi_write()** can be found in Appendix II, Sample Driver I. Any messages sent to **lpi_write()** will appear on the built-in Thinkjet printer.

There are times when **lpi_write()** is not sufficient to debug a driver. This can happen when the driver needs to operate very fast and the Thinkjet printer slows the driver down too much. It would also be unusable if you were rewriting a driver for the Thinkjet printer itself. In these cases you can send output to a serial card in slot B. You need to hook up a terminal to the serial interface in order to display the output. Terminals with built-in printers are most useful. Another IPC running the **Datacomm** program or other terminal emulator would also prove to be useful. Output messages going to the serial interface can prove to be very fast. In order to effect this just look closely at the **char()** routine in **kprintf.c**. Insure that the characters to be output are sent to the **initconsole()** routine instead of **lpi_write()**. If you are not seeing any of the expected messages on the terminal insure that:

The messages are really going to the serial interface, not to the printer.

You have a working serial card in slot B.

You can use **echo, cat,** or **Datacomm** to correctly access the serial interface.

You have the proper RS-232C cable arrangement for your terminal connection.

Your terminal is set to 9600 baud.

You are really exercising expected code in your driver.

Beyond these few utilities you will find the usual amount of inspiration and brain sweat helpful.[5]

5. Author's note: When all normal debugging efforts prove fruitless, I highly recommend taking a hot shower. Many elusive bugs have been dissolved in the hot mist.

6. OS Utility Routines

The following pages describe OS utilities which the driver may call. Many of these routines aid the driver by moving data between the kernel address map and the user program's address map. Some of the utility routines look at **u.u_segflg** before moving data. The **u.u_segflg** variable indicates whether the caller is a kernel routine, or a user routine. The utility routine then will move the data from or to the appropriate address map.

This section describes the typically needed system routines. There are many OS routines that are not documented by this section. It is beyond the scope of this document to describe all the internal OS routines.

blt - block copy routine

NAME

blt - copy a block of bytes as fast as possible

SYNOPSIS

```
blt (to_addr, from_addr, nbytes)
char *to_addr, *from_addr;
int nbytes;
```

DESCRIPTION

The **blt()** routine copies **nbytes** from **from_addr** to **to_addr.** If both addresses start on even word addresses, the transfer can be very quick. The IPC has been timed to transfer a large number of bytes at rates up to 750 Kbytes per second using the **blt()** routine.

The **blt()** routine does *not* do any address transformation. Should this be necessary, the transformation should be done to an address before handing the address to **blt()**.

NOTE

This routine is supplied in the library libc.os.a. blt() is an artifact from the Release 1.0.0 C compiler. To be precise, libc.os.a is a version of the Release 5.0 libc.a compiled in system mode (no stack checking), and then a version of blt.o, also compiled in system mode, was appended to this library. **memcpy** is the functional equivalent of **blt()**, but **memcpy()** is not as fast.

BUGS

If a driver hands **blt()** a bogus address, and a supervisor mode bus error results, the system will panic.

copyin - copy bytes from user

NAME

copyin, copyout - copy bytes from/to user address map

SYNOPSIS

```
copyin (user_from, to, nbytes)
char *to, *user_from;
int nbytes;

copyout (from, user_to, nbytes)
char *user_to, *from;
int nbytes;
```

DESCRIPTION

copyin() copies **nbytes** from the address **user_from** to address **to**, and **copyout()** copies **nbytes** to the address **user_to** in the user memory data map from the address **from**. Please note that **p_iomove()** is the recommended routine to use for copying structures when handling ioctl calls in a driver, as **p_iomove()** is sensitive to **u.u_segflg**.

DIAGNOSTICS

Returns -1 if a bus error occurs during the copy operation.

EXAMPLE

The following examples show how these routines could be used. They show how a tty driver could move a structure to and from the user data map using **copyin()** and **copyout()**.

```
...
struct sgttyb tb;
...
case TIOCSETP:
      if (copyin (arg, &tb, sizeof(tb) ))
            {
            u.u_error = EFAULT;
            break;
            }
      ...
case TIOCGETP:
      ...
      if (copyout (&tb, arg, sizeof(tb) ))
            {
            u.u_error = EFAULT;
            break;
            }
```

Note that the routines set an error condition in **u.u_error** if a bus error occurs when the structure is copied.

copyiin - copy bytes from user

NAME

copyiin, copyiout - copy bytes from/to user instruction map

SYNOPSIS

```
int
copyiin (user_from, to, nbytes)
char *to, *user_from;
int nbytes;

int
copyiout (from, user_to, nbytes)
char *user_to, *from;
int nbytes;
```

DESCRIPTION

copyiin() copies **nbytes** from the address **user_from** to address **to**, and **copyiout()** copies **nbytes** to the address **user_to** in the user instruction memory map from the address **from**.

DIAGNOSTICS

Returns -1 if a bus error occurs during the copy operation.

cpass - pass bytes back to user

NAME

cpass, passc - pass bytes to/from user

SYNOPSIS

```
unsigned char
passc (c)
char c;

unsigned char
cpass ()
```

DESCRIPTION

passc() and **cpass()** are sensitive to the setting of **u.u_segflg**. This means that the value of **u.u_segflg** will determine which memory map the routines read from or write to. The memory map accessed is shown below:

u.u_seglfg	map
0	User data map
1	Kernel data map
2	User instruction map

passc() writes **c** to the user's address map at address **u.u_base**. **passc()** updates **u.u_base**, **u.u_count**, and **u.u_offset**. These variables are normally set by the OS read routine according to the parameters passed by the user with the read call. **passc()** returns -1 when the last byte requested by the read call is transferred.

cpass() reads **c** from the user address map at **u.u_base**. **cpass()** updates **u.u_base**, **u.u_count**, and **u.u_offset**. These variables are normally set by the OS write routine according to the parameters passed by the user with the write call. **cpass()** returns -1 when the last byte indicated by the write call is obtained.

DIAGNOSTICS

Returns -1 if a bus error occurs on the transfer to or from the user memory map.

EXAMPLE

The following code fragment shows how the write handler for a theoretical driver could obtain characters using **cpass()**. The entry **hpwrite** would be called on a write to its associated special file in **/dev**. The example shown loops passing the characters returned by **cpass()** to **lpcanon()** until all the characters specified by the write are obtained.

```
hpwrite (dev)
dev_t dev;
{
        unsigned char c;
        while ((c=cpass()) >= 0)
                lpcanon (c);
}
```

dopen - kernel call to a driver open routine

NAME

dopen - kernel call to a driver open routine

SYNOPSIS

```
int
dopen (dev, mode)
long dev;
int mode;
```

DESCRIPTION

dopen() is useful in calling the open routine associated with a character driver in the system. In particular, one driver may call another driver's open routine to facilitate using the system resources.

dev is the major device number of the driver to be called. Specifically, **dev** shifted right 24 bits yields the index into the **cdevsw** table. **dev** unshifted is passed as the first argument to the driver open routine. **mode** is the second argument passed to the driver open routine.

dopen() handles the needed housekeeping to maintain the integrity of the kernel data structures. **dopen()** first saves the current contents of **u.u_segflg** and **u.u_error**. **dopen()** then sets **u.u_segflg** to 1 to indicate to the driver being called that its data will be in the kernel data map. **u.u_error** is then set to zero. The driver open routine is then called. When the driver returns **u.u_segflg** and **u.u_error** are restored, and **dopen()** returns. The error status returned by the called open routine, found in **u.u_error** after the other driver open routine returns, is the return value of **dopen()**.

EXAMPLE

In order to call the open routine of the built-in HPIB character driver, which is found in the 19th slot of the **cdevsw** table, you would use:

```
dopen ( (19<<24) ,O_RDWR);
```

This allows that the first entry in the **cdevsw** table is entry number zero. Note also that O_RDWR is defined in the include file **fcntl.h**.

dclose - kernel close call to a character driver

NAME

dclose - kernel close call to a character driver

SYNOPSIS

```
int
dclose (dev)
long dev;
```

DESCRIPTION

dclose() is useful in calling the close routine associated with a character driver in the system. In particular, one driver may call another driver's close routine to facilitate using the system resources.

dev is the major device number of the driver to be called. Specifically, **dev** shifted right 24 bits yields the index into the **cdevsw** table. **dev** unshifted is passed as the first argument to the driver close routine.

dclose() handles the needed housekeeping to maintain the integrity of the kernel data structures. **dclose()** first saves the current contents of **u.u_segflg** and **u.u_error**. **dclose()** then sets **u.u_segflg** to 1 to indicate to the driver being called that its data will be in the kernel data map. **u.u_error** is then set to zero. The driver close routine is then called. When the driver returns **u.u_segflg** and **u.u_error** are restored, and **dclose()** returns. The error status returned by the called close routine, found in **u.u_error** after the other driver close routine returns, is the return value of **dclose()**.

EXAMPLE

In order to call the close routine of the built-in printer character driver, associated with **/dev/lp**, which is found in the 7th slot of the **cdevsw** table, you would use:

```
dclose (7<<24);
```

This allows that the first entry in the **cdevsw** table is entry number zero.

dwrite, dread - kernel write/read call to a character driver

NAME

dwrite, dread - kernel write/read call to a character driver

SYNOPSIS

```
int
dwrite (dev, base, count, offset)
long dev;
caddr_t base;
unsigned int count;
off_t offset;

int
dread (dev, base, count, offset)
long dev;
caddr_t base;
unsigned int count;
off_t offset;
```

DESCRIPTION

dwrite() and **dread()** are useful in calling the write/read routine associated with a character driver in the system. In particular, one driver may call another driver's write/read routine to facilitate using the system resources.

dev is the major device number of the driver to be called. Specifically, **dev** shifted right 24 bits yields the index into the **cdevsw** table. **dev** unshifted is passed as the first argument to the driver write/read routine. **base, count,** and **offset** are the new arguments to pass to the driver write/read routine respectively through the kernel data elements **u.u_base, u.u_count, u.u_offset.**

dwrite() and **dread()** handle the needed housekeeping to maintain the integrity of the kernel data structures. **dwrite()** and **dread()** first save the current contents of **u.u_segflg, u.u_error, u.u_base, u.u_count,** and **u.u_offset.** They then set **u.u_segflg** to 1 to indicate to the driver being called that its data will be in the kernel data map. **u.u_error** is then set to zero. Also **u.u_base, u.u_count,** and **u.u_offset** are respectively set to **base, count,** and **offset.** The driver write/read routine is then called. When the driver returns **u.u_segflg, u.u_error, u.u_base, u.u_count,** and **u.u_offset** are restored, and **dwrite()** or **dread()** return. The error status returned by the called write/read routine, found in **u.u_error,** becomes the return value of **dwrite()** or **dread().**

EXAMPLE

In order to call the write routine of the terminal0 character driver, which is found in the 14th slot of the **cdevsw** table, you could use code somewhat like the following fragment.

```
unsigned char charray [BUFSIZE];
register int size;
.
.
.
while (u.u_count > 0)
    {
    if (u.u_count > BUFSIZE)
        size = BUFSIZE;
    else
        size = u.u_count;

    if (p_iomove (charray, u.u_base, size, FREAD) == -1)
        {
        u.u_error = EFAULT;
        break;
        }

    u.u_count -= size;
    u.u_base += size;

    /*
     * Now call dwrite()
     */
    error = dwrite ( (14<<24), charray, size, 0);

    if (error != 0)
        {
        u.u_error = error;
        break;
        }
    .
    .
    .
    }
```

dioctl - kernel ioctl call to a character driver

NAME

dioctl - kernel ioctl call to a character driver

SYNOPSIS

```
int
dioctl (dev, cmd, arg)
long dev;
int cmd;
int *arg;
```

DESCRIPTION

dioctl() is useful in calling the ioctl routine associated with a character driver in the system. In particular, one driver may call another driver's ioctl routine to facilitate using the system resources.

dev is the major device number of the driver to be called. Specifically, **dev** shifted right 24 bits yields the index into the **cdevsw** table. **dev** unshifted is passed as the first argument to the driver ioctl routine.

dioctl() handles the needed housekeeping to maintain the integrity of the kernel data structures. **dioctl()** first saves the current contents of **u.u_segflg** and **u.u_error**. **dioctl()** then sets **u.u_segflg** to 1 to indicate to the driver being called that its data will be in the kernel data map. **u.u_error** is then set to zero. The driver ioctl routine is then called. **cmd** is passed as the second argument to the driver ioctl routine. **arg** is passed as the third argument. When the driver returns **u.u_segflg** and **u.u_error** are restored, and **dioctl()** returns. The error status returned by the called ioctl routine, found in **u.u_error**, becomes the return value of **dioctl()**.

fubyte - fetch user byte

NAME

fubyte, fuibyte - fetch user byte

SYNOPSIS

```
char
fubyte (addr)
char *addr;

char
fuibyte (addr)
char *addr;
```

DESCRIPTION

fubyte() and **fuibyte()** return the byte pointed to by **addr** from the user data or instruction memory maps, respectively. **addr** is a user logical (unmapped) address. The **fubyte()** and **fuibyte()** routines will perform an address map translation before fetching the byte from the user space.

DIAGNOSTICS

Returns -1 if a bus error occurs on the read of a user memory map.

fuword - fetch user word

NAME

fuword, fuiword - fetch user word

SYNOPSIS

```
int
fuword (addr)
int *addr;

int
fuiword (addr)
int *addr;
```

DESCRIPTION

fuword() and **fuiword()** return the **int** pointed to by **addr** from the user data or instruction memory maps, respectively. **addr** is a user logical (unmapped) address. The **fuword()** and **fuiword()** routines will perform an address map translation before fetching the **int** from the user space.

DIAGNOSTICS

Returns -1 if a bus error occurs on the read of the user memory map.

p_iomove - driver read/write utility

NAME

p_iomove - driver read/write utility

SYNOPSIS

```
p_iomove (cp, bp, n, flag)
caddr_t cp;
caddr_t bp;
int n;          /* byte count to move */
int flag;       /* FREAD | FWRITE */
```

DESCRIPTION

The **p_iomove()** routine is a utility which a driver may find convenient for handling reads and writes. The **flag** options are:

FREAD	Copy **n** bytes from **bp** to **cp**
FWRITE	Copy **n** bytes from **cp** to **bp**

The FREAD and FWRITE flags are defined in **/usr/include/sys/file.h**.

If **u.u_segflg** is:

0	**bp** points to the user data map
1	**bp** points to the kernel data map
2	**bp** points to the user instruction map

DIAGNOSTICS

Returns -1 if an error occurred on the copy.

physio - block driver raw I/O utility

NAME

physio - block driver raw I/O utility

SYNOPSIS

```
physio (strat, bp, dev, rw, command)
int (*strat)(); /* strategy routine to call */
struct buf *bp;
dev_t dev;
int rw;         /* 1 -> read, 0 -> write */
int command;
```

DESCRIPTION

The **physio()** routine provides common services for a block driver which supports *raw I/O*. When a block driver calls **physio()**, it will generally supply the address of its own strategy routine as the **strat** parameter. The **bp** parameter should be set to point to a local buffer header (e.g. "struct buf mybuf") declared in the driver. The **rw** parameter is or'ed with B_BUSY and B_PHYS, and then copied to **bp->b_flags**, to be passed to the strategy routine. The **command** parameter is copied to the **bp->command** variable.

physio() performs several services for the block driver. First, it checks the user's buffer to make sure the entire buffer can be accessed without incurring a bus error. Next, it sets up the **buf** parameters and calls the block driver's strategy routine. It then waits for the operation to finish. When the B_DONE bit is set in the **b_flags** variable for the block, it decrements the byte count left in the user request. If bytes are left to be transferred, it will loop calling the strategy routine until the request is finished. When the transfer is done, the routine checks to see if any errors occurred, and sets **u.u_error** appropriately.

physck - error check block transfer

NAME

physck - error check block transfer

SYNOPSIS

```
physck (nblocks, rw)
daddr_t nblocks;
int rw;        /* 1 -> read, 0 -> write */
```

DESCRIPTION

The **physck()** utility is useful to call before a block driver performs *raw access*. It first checks that the raw request is limited to an integral number of blocks. It then checks to make sure that the size of the request does not exceed the device capacity. To do this it adds the base address of the request, given by **u.u_offset**, to the size of the request, given by **u.u_count**, and compares this to the size of the device, passed in **nblocks**. If the request will go out of bounds, the ENXIO error is set, and the routine returns 0.

If all the tests pass, the routine returns a 1.

sleep, wakeup - control process activity

NAME

sleep, wakeup - control process activity

SYNOPSIS

```
sleep (channel, priority)
char *channel;
int priority;

wakeup (channel)
char *channel;
```

DESCRIPTION

Calling **sleep()** causes the process to be idled. All processes which may be sleeping on **channel** will be made ready to run when a **wakeup()** is issued. The value for **channel** is generally chosen to be a unique value for the given reason for sleeping; e.g. a buffer address or status variable. The reason for uniqueness is to avoid having two processes waiting on the same address for different reasons. If a wakeup occurs on the address for one reason, one process would be awakened by accident.

The **priority** parameter is used to decide if a signal (e.g. a Control-C issued from the keyboard) will cause the process to be awakened. If **priority** is less than PZERO, then the signal will *not* disturb the sleep. PZERO is defined in **sys/param.h** to equal 25. If **priority** is greater than PZERO, then the signal *will* wake up the process

No value is returned by **sleep()**.

BE CAREFUL!! The **sleep(3)** routine in **libc.a** is *not* the same as the **sleep()** routine in the OS ROMs. In order to call the **sleep()** routine in the ROMs, it may be best to use the jump table entry point: **_jt_sleep**. This will jump to the sleep routine in the ROMs. If you call **sleep**, you will likely pull the sleep module from the **libc.a** library, and this will not be what you want.

The **wakeup()** routine wakes up all processes which are sleeping on **channel**.

spl - set processor priority level

NAME

spl - set processor priority level

SYNOPSIS

```
int             int             int             int
spl0 ()         spl1 ()         spl2 ()         spl3 ()

int             int             int             int
spl4 ()         spl5 ()         spl6 ()         spl7 ()

int
splx (x)
int x;
```

DESCRIPTION

The **spl()** routines allow drivers to set the processor priority level to the level specified by the call. A call to **spl0()** allows all interrupts. A call to **spl7()** masks out all interrupts but the level 7 keyboard interrupt. All of the **spl0()** through **spl7()** routines return the current priority level.

The **splx(x)** routine can be used to set the priority level to level **x**. A common ploy for a routine which may be called at either an interrupt level or at program level, is to do the following:

```
int x;
...
x = spl5();
...
(critical section of code)
...
splx(x);
```

The above example shows the interrupt mask level being set to level 5. The current level is saved in variable **x**. When the critical section has completed, then the interrupt mask level is restored to its original value.

subyte - set user byte

NAME

subyte, suibyte - set user byte

SYNOPSIS

```
char
subyte (addr, c)
char *addr, c;

char
suibyte (addr, c)
char *addr, c;
```

DESCRIPTION

subyte() and **suibyte()** set the byte pointed to by **addr** equal to **c** from the user data or instruction memory maps, respectively. **addr** is a user logical (unmapped) address. The **subyte()** and **suibyte()** routines will perform an address map translation before writing the byte to the user space.

DIAGNOSTICS

Returns -1 if a bus error occurs on the write of the user memory map.

suword - set user word

NAME

suword, suiword - set user word

SYNOPSIS

```
int
suword (addr, w)
long *addr, w;

int
suiword (addr, w)
long *addr, w;
```

DESCRIPTION

suword() and **suiword()** set the long pointed to by **addr** equal to **w** in the user data or instruction memory maps, respectively. **addr** is a user logical (unmapped) address. The **suword()** and **suiword()** routines will perform an address map translation before writing the word to the user space.

DIAGNOSTICS

Returns -1 if a bus error occurs on the write of the user memory map.

timeout - timeout utility

NAME

timeout - timeout utility

SYNOPSIS

```
timeout (fun, arg, tim)
int (*fun)();
caddr_t arg;
int tim;

remove_timeout (fun, arg);
int (*fun)();
caddr_t arg;
```

DESCRIPTION

The **timeout()** routine can be used to set a timeout. The **tim** parameter specifies the timeout duration in **tim * 100** milliseconds. Every system clock tick, the timeout handler is called to check if a timeout has elapsed. When the time has passed, the timeout utility will call the function specified by **(*fun)()**. The call to the routine will be made at interrupt level 1.

The **remove_timeout()** routine is available to remove timeouts. The routine scans the timeout array in the OS looking for an entry which matches the **fun** and **arg** parameters passed in. When it finds a match, it removes the timeout from the table. Interrupts are disabled during the search and removal of timeouts.

BUGS

Space in the timeout table is available for only 20 timeouts. Should the OS run out of available slots, it will panic (soft boot). Therefore be sure that your timeout routine has finished before starting a new one.

useracc - user access error checking

NAME

useracc - user access error checking

SYNOPSIS

```
useracc (user_ptr, length, access)
char *user_ptr;
int length;
int access;
```

DESCRIPTION

The **useracc()** routine is available to check a user region for accessability. The **access** parameter determines the type of access to be attempted to the region:

0	region is to be read by the driver.
1	region is to be written to by the driver.

The **user_ptr** parameter is a user map pointer supplied by a user program. The **useracc()** routine first converts the **user_ptr** to a supervisor mode pointer. It then attempts to read or write a byte at the address. If the operation succeeds, the routine then adds **length** to the pointer, and attempts to access the end of the region.

DIAGNOSTICS

The routine returns a 1 if the region is accessable. If a bus error occurs when the region is accessed, the routine returns 0.

vtop - user to supervisor pointer conversion

NAME

vtop - user to supervisor pointer conversion

SYNOPSIS

```
vtop (user_ptr)
char *user_ptr;
```

DESCRIPTION

The **vtop()** routine is a handy routine to convert a user mode pointer to a supervisor mode pointer. If a pointer is passed from a program to a driver which points to a data item in the user program, the driver must first convert the pointer to a supervisor mode pointer before it can use the pointer. To perform the conversion, the **vtop()** routine adds the base address of the process' data region in memory to **user_ptr**.

Appendix I - Post-Mortem Traceback Dumps

When a program dies, the IPC does not normally write a core file for diagnosis after the death. To assist in the diagnosis, the post-mortem dump facility should prove to be useful.

The post-mortem dump can be loaded into the OS after the IPC has powered on. The load is done by the driver loader program **dload**. The **PMroutine** contains a routine which knows how to print information to the internal Thinkjet printer. It also contains routines to print information about the cause of an OS crash and a traceback of routines called during execution leading up to the crash. The **PMroutine** will also print a post-mortem dump when user programs do something illegal. The **PMroutine** is loaded into the OS when the **Load_PM** program is run.

Load_PM is a script which in effect runs the command:

dload PMroutine

dload and **PMroutine** can be found on the Driver Writer's Utility Disc which accompanies this document.

To illustrate the benefit of the traceback, let us examine the traceback resulting from the following program, **/dwg/Sources/PM.src/PMexample.c**:

```
/*
 * Example.c
 *   This routine is an example of a program which dies because
 * it incurs a bus error.  The program induces a bus error by
 * writing to address 0x5fffff, which is a location where there is
 * no memory.
 */
char mes1[] = "Hello from Post-Mortem Example routine.\n";
char mes2[] = "In the foo routine.\n";
char mes3[] = "In the bar routine.\n";
char mes4[] = "This message should not be printed.\n";
main()
        {
        write(1, mes1, strlen(mes1));
        foo(1,2,3);
        }

foo(a,b,c)
int a,b,c;
        {
        write(1, mes2, strlen(mes2));
        bar(16,17,18);
        }

bar(a,b,c)
int a,b,c;
        {
        char *cp = (char *) 0x5fffff;
        write(1, mes3, strlen(mes3));
        *cp = 1;
        write(1, mes4, strlen(mes4));
        }
```

When this program is run, and the **PMroutine** has been loaded into your machine, then the following post-mortem will be printed out on your IPC's internal printer.

```
trap ... process bus error.
This process tried to touch non-existent memory.
The error occurred on a WRITE of data at address 0x5FFFFF
The pc was 0x2130 when the error occurred.

pid = XX sr = 0x0
 D0 0x1       D1 0x64      D2 0x0       D3 0x0
 D4 0x0       D5 0x0       D6 0x0       D7 0x0
 A0 0x5FFFFF  A1 0x7EE1C   A2 0x0       A3 0x0
 A4 0x0       A5 0x0       A6 0x7EDE0   A7 0x7EDDC

Now print a traceback of routine calls before the error.
  The numbers are all in hex. The arguments printed may
  or may not be valid as one cannot tell which bytes on
  the stack are arguments versus temporary variables.

...called by 0x20E4 possible args: [10] [11] [12]
...called by 0x209C possible args: [1] [2] [3]
...called by 0x204A possible args: [1] [7EE1C] [7EE24]
```

In analysing the data from the post-mortem traceback, you first need a list of where the routines in your program are located. The **nm(1)** utility is useful for obtaining this data. The following table shows the list of routines and their locations for the example program. It was extracted by issuing the command **nm -hn Example.**

```
0x00002000 T  start          0x0008002A D  _mes2
0x0000205C T  _main          0x00080040 D  _mes3
0x000020A4 T  _foo           0x00080056 D  _mes4
0x000020EC T  _bar           0x0008007C B  _argc_value
0x00002158 T  _strlen        0x0008007C D  _edata
0x0000218C T  _exit          0x00080080 B  _argv_value
0x000021A8 T  __cleanup      0x00080084 B  _environ
0x000021B4 T  __exit         0x00080088 B  float_soft
0x000021C0 T  _write         0x0008008C B  _errno
0x000021D4 T  __cerror       0x00080090 B  _errnet
0x000021EC T  _etext         0x00080094 B  _end
0x00080000 D  _mes1          0x00FF9800 A  float_loc
```

Now by referencing this address list, you can determine which routine incurred the error. The **PMroutine** reported that the **pc** was at 0x2130 when the error occurred. This address is between **_bar** and **_strlen** in the list, so the error occurred within the **bar()** routine. In the traceback printout, it was reported that **bar** was called by 0x20E4. This address falls between **_foo** and **_bar**, so the call was made by the **foo()** routine. Also the traceback shows that the possible args passed to **foo** are 10, 11, and 12 in hex. Looking back at the example code, these arguments are correctly reported. Next, the traceback shows that **foo** was called by 0x209C. This address falls between **_main** and **_foo**, so the call was made by the **main()** routine. The arguments shown are 1, 2, and 3.

Finally we see that **main** was called by 0x204A, which is in the **start** routine. This routine is automatically prepended to a program by the C compiler. The **start** routine is responsible for setting up the standard parameters to pass to the main routine. These

parameters are traditionally called **argc**, **argv**, and **environ**. The **argc** parameter contains the number of command line arguments present in the command line. The traceback shows its value as 1, which shows that the example routine was invoked using only its name. This is valid as all programs are passed their invocation name in the **argv** array. The **argv** parameter points to an array of character pointers, which in turn point to the command line arguments. The value passed is 0x7EE1C, which is an address of a word on the program's stack. The **environ** parameter points to an array of character pointers, which in turn point to environment lines passed by the program which **exec'ed** the program. The value is 0x7EE24, which is also an address of a word on the program's stack.

Appendix II - Sample Driver I.

The first example driver is a very simple one which defines the five entry points for a character device driver. When each entry point is called the driver prints an identifying message on the internal printer. The driver then returns.

The driver prints its message by calling the **lpi_write** OS routine in ROM. This routine prints the character passed to it on the internal printer.

```
/*
 * driver1.c
 *
 *   This is a sample driver for the Integral Personal Computer (IPC).
 * The driver simply prints a "here I am" message when its
 * open, close, read, write, and ioctl entry points are called.
 * The messages are printed to the built-in printer.
 *
 */

#include <sys/param.h>
#include <sys/dir.h>
#include <sys/ino.h>
#include <sys/inode.h>
#include <sys/user.h>

/*
 * the patch entries are scanned by the loader, dload, as it
 * loads the driver into the OS.  The patch entries specify
 * which routines in the driver are to be inserted, in this case,
 * into the cdevsw table in the OS.
 */
char *patch[] =
        {
        "cdevsw.name:driver1",
        "cdevsw.open<-driver1_open",
        "cdevsw.close<-driver1_close",
        "cdevsw.read<-driver1_read",
        "cdevsw.write<-driver1_write",
        "cdevsw.ioctl<-driver1_ioctl",
        0,
        };

char open_mes[] = "driver1: open routine called.\n";
char close_mes[] = "driver1: close routine called.\n";
char write_mes[] = "driver1: write routine called.\n";
char read_mes[] = "driver1: read routine called.\n";
char ioctl_mes[] = "driver1: ioctl routine called.\n";

driver1_open()
        {
        driver1_print( open_mes );
        }

driver1_close()
```

```
        {
        driver1_print( close_mes );
        }

driver1_read()
        {
        driver1_print( read_mes );
        }

driver1_write()
        {
        driver1_print( write_mes );
        u.u_count = 0;
        }

driver1_ioctl()
        {
        driver1_print( ioctl_mes );
        }

/*
 * write a string to the built-in printer.  The lpi_write routine
 * is an OS ROM routine which writes characters to the printer.
 * Expand the linefeed (\n) by adding a carriage return (\r).
 */
driver1_print (mes)
char *mes;
{
        for ( ; *mes; mes++)
                {
                lpi_write(*mes, 1);
                if (*mes == '\n')
                        lpi_write('\r', 1);
                }
}
```

The driver can be loaded by typing into PAM **dload -v driver1**.[6] The **-v** command line option for **dload** requests *verbose mode*. Verbose mode provides optional information about how the driver was loaded into the system. The expected output from invoking **dload -v driver1**, on an IPC using HP-UX release 5.0 without having any other drivers loaded into the system should look something like:

6. Note that in order for dload to work properly that the hp-ux.X.X.X symbol file corresponding to the OS version being used must be in your normal path search. Also, of course, dload must be in your path.

```
dload V1.5
Symbol table file is /usr/hp-ux.5.0.0
Loading driver1 (from driver1).

size of driver1 is 552 bytes.
Base address of allocated OS ram: 0xeefd00
Address of cdevsw table: 0xf061b2
Address of bdevsw table: 0xf060d2
Address of interrupt table: 0xf0646c
OS address of the buffer is 0xeefd00
Base address of proc: 0xdea000
  Address to clear memory (relative to proc): 0x180500
   _lpi_write: value 0x38d24
   _u: value 0xf1bf00
match for _driver1_ioctl at 0x68
match for _driver1_write at 0x48
match for _driver1_read at 0x30
match for _driver1_close at 0x18
match for _driver1_open at 0x0

make character device: /dev/driver1 number 0x19000000
Set cdevsw open (25) at 0xf063a6 to _driver1_open (0xeefd00)
Set cdevsw close (25) at 0xf063aa to _driver1_close (0xeefd18)
Set cdevsw read (25) at 0xf063ae to _driver1_read (0xeefd30)
Set cdevsw write (25) at 0xf063b2 to _driver1_write (0xeefd48)
Set cdevsw ioctl (25) at 0xf063b6 to _driver1_ioctl (0xeefd68)
driver1 is ready for use.
```

To demonstrate that it has been loaded, first examine the **/dev** directory. The **driver1** device file should be present in the directory. Try the command **view /dev/driver1**. This command will cause **PAM** to try to read from the driver. What will printed on the internal printer will be:

```
driver1: open routine called.
driver1: read routine called.
driver1: close routine called.
```

The printout shows that the driver's open routine, then the read routine, then the close routine were called. Now try to write to the driver. This can easily be done by giving the command **echo foo > /dev/driver1**. The printout which should result from this line is:

```
driver1: open routine called.
driver1: close routine called.
driver1: open routine called.
driver1: write routine called.
driver1: write routine called.
driver1: write routine called.
driver1: close routine called.
```